MMF 2021 FINAL EXAM 30 November 2018.

This is a **closed-book exam**: no books, no notes, no calculators, no phones, no tablets, no computers (of any kind) allowed.

Do **NOT** turn this page over until you are **TOLD** to start.

Duration of the exam: 3 hours.

Write your answers in the exam booklets provided.

Please fill-in **ALL** the information requested on the front cover of **EACH** exam booklet that you use.

The exam consists of 8 pages, including this one. Make sure you have all 8 pages.

The exam consists of 4 questions. **Answer all 4 questions**. The mark for each question is listed at the start of the question.

The exam was written with the intention that you would have ample time to complete it. You will be rewarded for concise well-thought-out answers, rather than long rambling ones. **We seek quality rather than quantity**.

Moreover, an answer that contains relevant and correct information as well as irrelevant or incorrect information will be awarded fewer marks than one that contains the same relevant and correct information only.

Write legibly. Unreadable answers are worthless.

1. [10 marks: 5 marks for each part]

Walter was having trouble debugging his program. He traced the problem to a certain section of his code, but what he was computing there was fairly complex. So, he decided to try a simpler example of what he thought might be wrong with his code to see if that might help him determine the problem.

Walter knew that, if

$$f(x) = \frac{e^x - 1}{x}$$

then

$$\lim_{x \to 0} f(x) = 1 \tag{1}$$

(For this question, just accept (1) as being true: you don't have to prove it.)

So, he expected that, if he computed $f(x)$ for smaller and smaller positive values of x, the computed values of $f(x)$ would get closer and closer to 1. He decided to test this conjecture, since he knew that odd things often happen in floating-point computation. So, he wrote a little MatLab program that computes $f(x)$ in IEEE double-precision floating-point arithmetic for $x = 10^{-k}$ and $k = 1, 2, \ldots, 15$. To his surprise, he got the results shown in the third column of the table on page 3.

He showed his results to his colleague, Irene, who suggested that he try computing instead

$$g(x) = \frac{e^x - 1}{\ln(e^x)}$$

where ln is the natural logarithm (also referred to as the logarithm to the base e (i.e., $\log_e$)).

Walter thought that this was a ridiculous suggestion, since $\ln(e^x) = x$, whence $f(x) = g(x)$ for all $x \in \mathbb{R}$ (assuming you define $f(0) = g(0) = 1$). Nevertheless, he tried Irene's suggestion and, to his surprise, he got the results shown in the fourth column of the table on page 3.

k	x	$f(x)$	$g(x)$
1	10^{-1}	1.051709180756477	1.051709180756476
2	10^{-2}	1.005016708416795	1.005016708416806
3	10^{-3}	1.000500166708385	1.000500166708342
4	10^{-4}	1.000050001667141	1.000050001666708
5	10^{-5}	1.000005000006965	1.000005000016667
6	10^{-6}	1.000000499962184	1.000000500000167
7	10^{-7}	1.000000049433680	1.000000050000002
8	10^{-8}	0.999999993922529	1.000000005000000
9	10^{-9}	1.000000082740371	1.000000000500000
10	10^{-10}	1.000000082740371	1.000000000050000
11	10^{-11}	1.000000082740371	1.000000000005000
12	10^{-12}	1.000088900582341	1.000000000000500
13	10^{-13}	0.999200722162641	1.000000000000050
14	10^{-14}	0.999200722162641	1.000000000000005
15	10^{-15}	1.110223024625157	1.000000000000000

(a) Explain why, when $f(x)$ is computed in IEEE double-precision floating-point arithmetic, the computed values first appear to be converging to 1 for $k = 1, 2, \ldots, 8$, but then diverge from 1 for $k = 11, 12, \ldots, 15$.

(b) Explain why, when $g(x)$ is computed in IEEE double-precision floating-point arithmetic, the computed values appear to be converging to 1 for $k = 1, 2, \ldots, 15$. In particular, explain why the computed values for $g(x)$ are so much more accurate than the computed values for $f(x)$ for $k = 11, 12, \ldots, 15$.

In answering the questions above, you can assume that the MatLab function $\exp(x)$ computes an accurate approximation to e^x. In particular, you can assume that

$$\exp(x) = \mathrm{e}^x(1 + \delta_x)$$

where δ_x changes with x, but its magnitude is at most a few multiples of ϵ_{mach}. (I.e., $|\delta_x| \leq c\,\epsilon_{\text{mach}}$ for some c that is at most 2 or 3.)

You can make some other reasonable assumption about the accuracy of the MatLab ln function, but remember that $\ln(u)$ is ill-conditioned for u near 1. If you make an assumption about the accuracy of the MatLab ln function, be sure to state what your assumption is.

2. [10 marks: 5 marks for each part]

Suppose you want to generate a pseudo-random variable X having the pdf

$$f_n(x) = \frac{x^n \mathrm{e}^{-x}}{n!} \qquad \text{for } x \geq 0$$

where $n \geq 0$ is an integer. Assume throughout this question that n is given. (I.e., you don't have a choice of n — it is specified — and n doesn't change throughout the question.)

You can try to compute the CDF of the distribution

$$F_n(x) = \int_0^x f_n(t)\, dt \qquad \text{for } x \geq 0$$

but the expression for $F_n(x)$ is fairly long and complicated if n is not very small. Hence, it is fairly messy to used the Inverse Transform Method. So, let's try the Acceptance-Rejection Method with

$$g_{\lambda_n}(x) = \lambda_n \mathrm{e}^{-\lambda_n x} \qquad \text{for } x \geq 0$$

where $\lambda_n > 0$. Also, note that, since we require that $f_n(x)/g_{\lambda_n}(x)$ is bounded for all $x \geq 0$, we must have $\lambda_n < 1$. Hence, assume throughout this question that $\lambda_n \in (0, 1)$.

Since $g_{\lambda_n}(x)$ is the pdf of the exponential distribution, it follows from our discussion in class that we can easily generate a random variable Y having this distribution by first generating a uniform $[0, 1]$ random variable U (using, for example, the function `rand` in MatLab) and then setting

$$Y = \frac{-1}{\lambda_n} \ln(U)$$

where ln is the natural logarithm (also referred to as the logarithm to the base e (i.e., $\log_{\mathrm{e}}$)).

(a) In part (b) below, we will discuss how to choose λ_n to optimize the Acceptance-Rejection Method developed below based on $g_{\lambda_n}(x)$, but, for this part of the question, assume only that you are given a $\lambda_n \in (0, 1)$.

For the Acceptance-Rejection Method, you need a parameter c_{n,λ_n} such that

$$f_n(x)/g_{\lambda_n}(x) \leq c_{n,\lambda_n} \tag{2}$$

for all $x \geq 0$.

- For a given integer $n \geq 0$ and a given $\lambda_n \in (0, 1)$, what is the best choice for the value of the parameter c_{n,λ_n}?
- State why to think the value for c_{n,λ_n} you chose is the best choice for the given $n \geq 0$ and the given $\lambda_n \in (0, 1)$.
- Write the Acceptance-Rejection Method in enough detail that a programmer who does not know any Mathematical Finance can program your version of the the Acceptance-Rejection Method to generate a pseudo-random variable X having the pdf $f_n(x)$ given above.

In writing your Acceptance-Rejection Method, you can use any of the standard MatLab functions, such as `rand`.

(b) For a given integer $n \geq 0$, what is the best choice of the parameter $\lambda_n \in (0,1)$ for your Acceptance-Rejection Method described in part (a)?

Given your choice of $\lambda_n \in (0,1)$, how efficient is your Acceptance-Rejection Method described in part (a)? Consider the cases for which

- n is fairly small (e.g., $1 \leq n \leq 5$)
- n is of medium size (e.g., $6 \leq n \leq 20$)
- n is large (e.g., $n > 20$)

In answering this question, you can use the table of values below for

$$\alpha_n = \frac{e^{-n}\,(n+1)^{n+1}}{n!}$$

n	α_n
1	1.4715
2	1.8270
3	2.1242
4	2.3848
5	2.6197
6	2.8352
7	3.0355
8	3.2233
9	3.4008
10	3.5695
11	3.7306
12	3.8850
13	4.0335
14	4.1767
15	4.3152
16	4.4494
17	4.5796
18	4.7063
19	4.8296
20	4.9498

as well as the asymptotic result that

$$\alpha_n \approx 1.1\sqrt{n}$$

for large n.

3. [10 marks]

Consider a *basket option* that is based on two underlyings, $S_t^{(1)}$ and $S_t^{(2)}$, with payoff at expiry (i.e., at time $t = T$) given by

$$h(S_T^{(1)}, S_T^{(2)}) = \max(\omega_1 S_T^{(1)} + \omega_2 S_T^{(2)} - \hat{K}, 0)$$

where ω_1 and ω_2 are real constants satisfying $\omega_1 \in [0,1]$, $\omega_2 \in [0,1]$, $\omega_1 + \omega_2 = 1$ and $\hat{K} \in \mathbb{R}$ is a positive constant. Assume that the two underlyings, $S_t^{(1)}$ and $S_t^{(2)}$, start with values $S_0^{(1)}$ and $S_0^{(2)}$, respectively, at time $t = 0$ and evolve in time according to the SDEs

$$\begin{aligned} dS_t^{(1)} &= rS_t^{(1)}dt + \sigma_1 S_t^{(1)} dW_t^{(1)} \\ dS_t^{(2)} &= rS_t^{(2)}dt + \sigma_2 S_t^{(2)} dW_t^{(2)} \end{aligned}$$

where r is the risk free interest rate, σ_1 and σ_2 are the volatilities associated with $S_t^{(1)}$ and $S_t^{(2)}$, respectively, and the Brownian motions, $W_t^{(1)}$ and $W_t^{(2)}$, are correlated with correlation coefficient $\rho \in [-1, 1]$. Hence,

$$\begin{aligned} S_T^{(1)} &= S_0^{(1)} \mathrm{e}^{(r-\sigma_1^2/2)T + \sigma_1 W_T^{(1)}} \\ S_T^{(2)} &= S_0^{(2)} \mathrm{e}^{(r-\sigma_2^2/2)T + \sigma_2 W_T^{(2)}} \end{aligned}$$

and

$$\begin{aligned} W_T^{(1)} &= \sqrt{T}\Big(\sqrt{1-\rho^2} Z^{(1)} + \rho Z^{(2)}\Big) \\ W_T^{(2)} &= \sqrt{T} Z^{(2)} \end{aligned}$$

where $Z^{(1)} \sim N(0,1)$, $Z^{(2)} \sim N(0,1)$ and $Z^{(1)}$ and $Z^{(2)}$ are independent.

The price of this option at time $t = 0$ is

$$P_0 = \mathbb{E}[\mathrm{e}^{-rT} h(S_T^{(1)}, S_T^{(2)})]$$

We can easily do a "standard" Monte Carlo simulation to approximate P_0 as follows.

(a) For $n = 1, 2, \ldots, N$, let

$$Y_n = \mathrm{e}^{-rT} h(S_{T,n}^{(1)}, S_{T,n}^{(2)})$$

where

$$\begin{aligned} S_{T,n}^{(1)} &= S_0^{(1)} \mathrm{e}^{(r-\sigma_1^2/2)T + \sigma_1 W_{T,n}^{(1)}} \\ S_{T,n}^{(2)} &= S_0^{(2)} \mathrm{e}^{(r-\sigma_2^2/2)T + \sigma_2 W_{T,n}^{(2)}} \end{aligned}$$

and

$$\begin{aligned} W_{T,n}^{(1)} &= \sqrt{T}\Big(\sqrt{1-\rho^2} Z_n^{(1)} + \rho Z_n^{(2)}\Big) \\ W_{T,n}^{(2)} &= \sqrt{T} Z_n^{(2)} \end{aligned}$$

and each $Z_n^{(1)} \sim N(0,1)$, each $Z_n^{(2)} \sim N(0,1)$ and all the $Z_n^{(1)}$ and $Z_n^{(2)}$, for $n = 1, 2, \ldots, N$, are independent.

(b) Approximate the option price P_0 by

$$\hat{P}_0 = \frac{1}{N} \sum_{n=1}^{N} Y_n$$

Assume that you have a function, such as `blsprice` in MatLab,

`[Call, Put] = blsprice(`S_0`,` K`,` r`,` T`,` σ`)`

that computes the price at time $t = 0$ of a "vanilla" European call or put option that expires at time $t = T$, with strike price K, risk free interest rate r, volatility σ and underlying S_t that starts with value S_0 at time $t = 0$ and evolves in time according to the SDE

$$dS_t = rS_t dt + \sigma S_t dW_t$$

Assume also that you have a program such as `randn` in MatLab that generates independent standard normal random numbers (i.e., generates independent $Z \sim N(0,1)$). (You can use any other standard MatLab functions as well.)

How can you use `blsprice` and `randn` together with conditional expectation to develop a more efficient Monte Carlo simulation than the "standard" one given above to approximate the price P_0 of this basket option?

The description of your new Monte Carlo simulation should be detailed enough so that someone who does not know any mathematical finance can implement it easily in MatLab.

4. [13 marks: 3 marks for part (a) and 5 marks for each of parts (b) and (c)]

We didn't have time to discuss Asian options in detail; I hope you talked about them in another course.

Here's a very quick summary. If S_t is the stock price that evolves according to the standard SDE

$$dS_t = rS_t dt + \sigma S_t dW_t$$

where r is the risk free interest rate, σ is the volatility and W_t is a standard Brownian motion, and you also let

$$I_t = \int_0^t S_\tau d\tau$$
$$A_t = I_t/t$$

then the price of an arithmetic-average European-style Asian option at time $t \leq T$ is $V(t, S, I)$, where $V(t, S, I)$ satisfies the PDE

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} + S\frac{\partial V}{\partial I} - rV = 0 \tag{3}$$

with an appropriate terminal condition at time $t = T$ based on the payoff of the option at the option expiry time $t = T$ and appropriate boundary conditions.

The PDE (3) is a fairly simple extension of the standard Black-Scholes PDE. However, one interesting thing about the PDE (3) is that it seems like a parabolic PDE in S (like the Black-Scholes PDE or the Heat Equation), but like a hyperbolic PDE in I (like the Transport Equation). This creates some challenges to solve (3) numerically.

To get some idea of how to solve these mixed parabolic-hyperbolic PDEs, consider the simpler PDE

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial y^2} \tag{4}$$

for $u(t, x, y)$, where $t \in (0, T)$, $x \in (0, 1)$ and $y \in (0, 1)$. Assume that you are given an initial condition at $t = 0$

$$u(0, x, y) = u_0(x, y) \quad \text{for } x \in [0, 1] \text{ and } y \in [0, 1]$$

and appropriate boundary conditions for $t \in (0, T]$.

(a) Give a numerical method to solve (4). You numerical method should be both consistent and stable.

(b) What is the order of consistency of your numerical method specified in part (a) to solve (4)? More specifically, state what the order of consistency is in t, x and y. Justify your answer.

(c) Show that your numerical method specified in part (a) to solve (4) is stable.